Stabilizing Maximal Independent Set in Unidirectional Networks is Hard

Toshimitsu Masuzawa, Sébastien Tixeuil

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Stabilizing Maximal Independent Set in Unidirectional Networks is Hard

Toshimitsu Masuzawa — Sébastien Tixeuil

N° 6880

Mars 2009

Thème NUM

apport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6880--FR+ENG

Stabilizing Maximal Independent Set in Unidirectional Networks is Hard

Toshimitsu Masuzawa* , Sébastien Tixeuil†

Thème NUM — Systèmes numériques
Projet Grand large

Rapport de recherche n° 6880 — Mars 2009 — 18 pages

Abstract: A distributed algorithm is self-stabilizing if after faults and attacks hit the system and place it in some arbitrary global state, the system recovers from this catastrophic situation without external intervention in finite time. In this paper, we consider the problem of constructing self-stabilizingly a *maximal independent set* in uniform unidirectional networks of arbitrary shape. On the negative side, we present evidence that in uniform networks, *deterministic* self-stabilization of this problem is *impossible*. Also, the *silence* property (*i.e.* having communication fixed from some point in every execution) is impossible to guarantee, either for deterministic or for probabilistic variants of protocols.

On the positive side, we present a deterministic protocol for networks with arbitrary unidirectional networks with unique identifiers that exhibits polynomial space and time complexity in asynchronous scheduling. We complement the study with probabilistic protocols for the uniform case: the first probabilistic protocol requires infinite memory but copes with asynchronous scheduling, while the second probabilistic protocol has polynomial space complexity but can only handle synchronous scheduling. Both probabilistic solutions have expected polynomial time complexity.

Key-words: Distributed systems, Distributed algorithm, Maximal Independent Set, Unidirectional Networks, Self-stabilization, Probabilistic self-stabilization

* Osaka University, Japan
† Université Pierre & Marie Curie - Paris 6, LIP6-CNRS & INRIA Grand Large, France

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

L'Auto-stabilisation d'un Ensemble Maximal Indépendant dans les Réseaux Unidirectionels est Difficile

Résumé : Un algorithme distribué est auto-stabilisant si après que des fautes et des attaques ont frappé le système et l'ont placé dans un état global arbitraire, le système récupère en temps fini un fonctionnement correct sans intervention extérieure. Dans cet article, nous considérons le problème de la construction auto-stabilisante d'un *ensemble maximal indépendant* dans des réseaux uniformes et unidirectionels quelconques. Nous présentons un résultat négatif qui indique que dans les réseaux uniformes, l'auto-stabilisation *déterministe* de ce problème est *impossible* à résoudre. De plus, la propriété de *silence* (*i.e.* garantir qu'à partir d'un point de chaque exécution, les communications entre les nœuds du réseau sont fixées) est impossible à garantir, tant pour les variantes deterministes que probabilistes des protocoles.

Nos résultats positifs sont multiples. Nous présentons un protocole déterministe pour les réseaux unidirectionels *identifiés* quelconques qui présente une complexité en temps et en espace qui reste polynomiale, avec un ordonnancement asynchrone. Nous complétons l'étude avec des protocoles probabilistes dans le cas uniforme : le premier protocole requiert une mémoire infinie mais supporte un ordonnancement asynchrone, le deuxième protocole utilise une mémoire polynomiale mais requiert un ordonnancement synchrone. Les deux protocoles ont une compexité moyenne en temps polynomiale.

Mots-clés : Systèmes distribués, Algorithme distribué, Ensemble Maximal Indépendant, Réseaux Unidirectionels, Auto-stabilisation, Auto-stabilisation probabiliste

1 Introduction

One of the most versatile technique to ensure forward recovery of distributed systems is that of *self-stabilization* [10, 11]. A distributed algorithm is self-stabilizing if after faults and attacks hit the system and place it in some arbitrary global state, the system recovers from this catastrophic situation without external (*e.g.* human) intervention in finite time.

The vast majority of self-stabilizing solutions in the literature [11] considers *bidirectional* communications capabilities, *i.e.* if a process u is able to send information to another process v, then v is always able to send information back to u. This assumption is valid in many cases, but can not capture the fact that asymmetric situations may occur, *e.g.* in wireless networks, it is possible that u is able to send information to v yet v can not send any information back to u (u may have a wider range antenna than v). Asymmetric situations, that we denote in the following under the term of *unidirectional* networks, preclude many common techniques in self-stabilization from being used, such as preserving local predicates (a process u may take an action that violates a predicate involving its outgoing neighbors without u knowing it, since u can not get any input from its outgoing neighbors).

Related works Self-stabilizing solutions are considered easier to implement in bidirectional networks since detecting incorrect situations requires less memory and computing power [3], recovering can be done locally [2], and Byzantine containment can be guaranteed [17, 18, 20].

Investigating the possibility of self-stabilization in unidirectional networks was recently emphasized in several papers [1, 6, 7, 8, 13, 14, 9, 15, 5]. However, topology or knowledge about the system varies: [7] considers *acyclic* unidirectional networks, where erroneous initial information may not loop; [1, 6, 9, 13] assume *unique identifiers* and *strongly connected* so that global communication can be implemented; [8, 14, 15] makes use of *distinguished processes* yet operate on arbitrary unidirectional networks.

Tackling arbitrary *uniform* unidirectional networks in the context of self-stabilization proved to be hard. In particular, [5, 4] studied the self-stabilizing vertex coloring problem in unidirectional uniform networks (where adjacent nodes must ultimately output different colors). Deterministic and probabilistic solutions to the vertex coloring problem [16, 19] in bidirectional networks have *local* complexity (Δ states per process are required, and $O(\Delta)$ –resp. $O(1)$– actions per process are needed to recover from arbitrary state in the case of a deterministic –resp. probabilistic– algorithm). By contrast, in unidirectional networks, [5] proves a lower bound of n states per process (where n is the network size) and a recovery time of at least $n(n-1)/2$ actions in total (and thus $\Omega(n)$ actions per process) in the case of deterministic uniform algorithms, while [4] provides a probabilistic solution that remains either local in space *or* local in time, but not both.

Our contribution In this paper, we consider the problem of constructing self-stabilizingly a *maximal independent set* in uniform unidirectional networks of arbitrary shape. It turns out that local maximization (*i.e.* maximal independent set) is strictly more difficult than

local predicate maintainance (*i.e.* vertex coloring). On the negative side, we present evidence that in uniform networks, *deterministic* self-stabilization of this problem is *impossible*. Also, the *silence* property (*i.e.* having communication fixed from some point in every execution) is impossible to guarantee, either for deterministic or for probabilistic variants of protocols.

On the positive side, we present a deterministic protocol for networks with arbitrary unidirectional networks with unique identifiers that exhibits $O(m \log n)$ space complexity and $O(D)$ time complexity in asynchronous scheduling, where n is the network size and D is the network diameter. We complement the study with probabilistic protocols for the uniform case: the first probabilistic protocol requires infinite memory but copes with asynchronous scheduling (stabilizing in time $O(\log n + \log \ell + D)$, where ℓ denotes the number of fake identifiers in the initial configuration), while the second probabilistic protocol has polynomial space complexity (in $O(m \log n)$) but can only handle synchronous scheduling (stabilizing in time $O((n + \ell) \log n)$).

Outline The remaining of the paper is organized as follows: Section 2 presents the programming model and problem specification. Section 3 presents our negative results, while Section 4 details the protocols. Section 5 gives some concluding remarks and open questions.

2 Preliminaries

Program model A program consists of a set V of n processes. A process maintains a set of variables that it can read or update, that define its *state*. A process contains a set of *constants* that it can read but not update. A binary relation E is defined over distinct processes such that $(i, j) \in E$ if and only if j can read the variables maintained by i; i is a *predecessor* of j, and j is a *successor* of i. The set of predecessors (resp. successors) of i is denoted by $P.i$ (resp. $S.i$), and the union of predecessors and successors of i is denoted by $N.i$, the *neighbors* of i. The *ancestors* of process i is recursively defined as follows: predecessors of i are ancestors of i, and ancestors of each predecessor of i are also ancestors of i. The *descendants* of i are similarly defined using successors (instead of predecessors).

For processes i and j in V, $d(i, j)$ denotes the *distance* (or the length of the shortest path) *from i to j* in the directed graph (V, E). We define, for convenience, the distance as $d(i, i) = 0$ and $d(i, j) = \infty$ if i is not reachable to j. The *diameter* D is defined as $D = \max\{d(i, j) \mid (i, j) \in V \times V, d(i, j) \neq \infty\}$.

An action has the form $\langle name \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$. A *guard* is a Boolean predicate over the variables of the process and its predecessors. A *command* is a sequence of statements assigning new values to the variables of the process. We refer to a variable v and an action a of process i as $v.i$ and $a.i$ respectively. A *parameter* is used to define a set of actions as one parameterized action. Notice that actions of a process are completely independent of its successors.

A *configuration* of the program is the assignment of a value to every variable of each process from its corresponding domain. Each process contains a set of actions. In some configuration, an action is *enabled* if its guard is **true** in the configuration, and a process

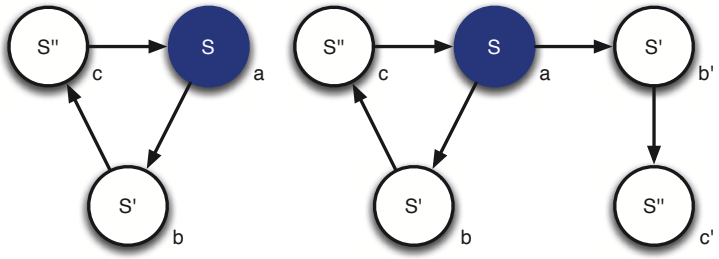
is *enabled* if it has at least one enabled action in the configuration. A *computation* is a maximal sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that for each configuration γ_i, the next configuration γ_{i+1} is obtained by executing the command of at least one action that is enabled in γ_i. Maximality of a computation means that the computation is infinite or it terminates in a configuration where none of the actions are enabled. A program that only has terminating computations is *silent*.

A *scheduler* is a predicate on computations, that is, a scheduler is a set of possible computations, such that every computation in this set satisfies the scheduler predicate. We consider only *weakly fair* schedulers, where no process can remain enabled in a computation without executing any action. We distinguish three particular schedulers in the sequel of the paper: the *distributed* scheduler corresponds to predicate **true** (that is, all weakly fair computations are allowed). The *locally central* scheduler implies that in any configuration belonging to a computation satisfying the scheduler, no two enabled actions are executed simultaneously on neighboring processes. The *synchronous* scheduler implies that in any configuration belonging to a computation satisfying the scheduler, every enabled process executes one of its enabled actions.

The distributed and locally central schedulers model *asynchronous* distributed systems. In asynchronous distributed systems, time is usually measured by *asynchronous rounds* (simply called *rounds*). Let $E = \gamma_0, \gamma_1, \ldots$ be a computation. The first round of E is the minimum prefix of E, $E_1 = \gamma_0, \gamma_1, \ldots, \gamma_k$, such that every enabled process in γ_0 executes its action or becomes disabled in E_1. Round t ($t \geq 2$) is defined recursively, by applying the above definition of the first round to $E' = \gamma_k, \gamma_{k+1}, \ldots$. Intuitively, every process has a chance to update its state in every round.

A configuration *conforms* to a predicate if this predicate is **true** in this configuration; otherwise the configuration *violates* the predicate. By this definition every configuration conforms to predicate **true** and none conforms to **false**. Let R and S be predicates over the configurations of the program. Predicate R is *closed* with respect to the program actions if every configuration of the computation that starts in a configuration conforming to R also conforms to R. Predicate R *converges* to S if R and S are closed and any computation starting from a configuration conforming to R contains a configuration conforming to S. The program *deterministically stabilizes* to R if and only if **true** converges to R. The program *probabilistically stabilizes* to R if and only if **true** converges to R with probability 1.

Problem specification Each process i defines a function $mis.i$ that takes as input the states of i and its predecessors, and outputs a value in $\{\mathbf{true}, \mathbf{false}\}$. The *unidirectional maximal independent set* (denoted by UMIS in the sequel) predicate is satisfied if and only if for every $i \in V$, either $mis.i = \mathbf{true} \wedge \forall j \in N.i, mis.j = \mathbf{false}$ or $mis.i = \mathbf{false} \wedge \exists j \in N.i, mis.j = \mathbf{true}$.



(a) System A (b) System B

Figure 1: Impossibility of self-stabilizing UMIS

3 Impossibility Results in anonymous networks

In this section, we consider anonymous and uniform networks, where processes of the same in-degree execute exactly the same code (note however that probabilistic protocols may exhibit different actual behaviors when making use of a random variable).

Theorem 1 *There exists no silent self-stabilizing solution for the UMIS problem.*

Proof. Assume there exists such a solution and consider System A as depicted in Figure 1.(a). Since the protocol is silent, it reaches a terminal configuration where exactly one of the three processes, says a, has $mis.a = \textbf{true}$. Now consider the system in Figure 1.(b), with the states of the processes in the tail (that is b' and c') being the same as those in the 3-cycle (that is b and c). Both processes with state S' (b and b') have the same in-degree and the same predecessor; as the one in the cycle (b) is silent, the second one (b') is also silent. Both processes with state S'' (c ad c') have the same in-degree and the same predecessor state; as the one in the cycle (c) is silent, the second one (c') is also silent. As a result, both processes b' and c' in the tail of System B never move. Since the UMIS function is based solely on the current state, in-degree, and predecessor state, the UMIS function returns the same result for both processes b and b' in state S' and for both processes c and c' in state S''. So, both processes b' and c' in the tail are not in the UMIS. Overall, System B describes a terminal configuration that is not a maximal independent set (the UMIS predicate does not hold at c'). □

Notice that the impossibility results of Theorem 1 holds even for probabilistic potential solutions. We now prove that relaxing the silence property still prevents the existence of deterministic solutions.

Theorem 2 *There exists no deterministic self-stabilizing solution for the UMIS problem.*

Proof. Assume there exists such a solution and consider the two systems A and B that are depicted in Figure 1. We consider a computation of system A, that eventually

ends up in a stable output of the *mis* function for all processes a, b, and c (a being the one process with $mis.a = \textbf{true}$), and construct a sibling execution in System B as follows:

- processes b and b' (resp. c and c') in System B have the same initial states as b (resp. c) in System A,
- anytime process b (resp. c) is executed in System A, both processes b and b' (resp. c and c') are executed in System B,
- anytime a is executed in System A, a is also executed in System B.

Now, at any time, in System B, both processes b and b' are in the same state, with the same predecessors' states. As a result, the output of their *mis* function is the same. The same holds for processes c and c'. Since System A eventually ends up in a configuration from which all *mis* functions are stable, the same holds for system B, where $mis.b'$ and $mis.c'$ both return **false**. As a result, a UMIS is never constructed in System B. □

4 Possibility Results

The previous impossibility results yield that for the deterministic case, only non uniform networks admit a self-stabilizing solution for the UMIS problem. In section 4.1, we present such a deterministic solution.

For anonymous and uniform networks, there remains the probabilistic case. We proved that probabilistic yet silent solutions are impossible, so both our solutions are non-silent. The one that is presented in Section 4.2 performs in asynchronous networks but requires unbounded memory, while the one that is presented in Section 4.3 performs in synchronous networks and uses $O(m \log n)$ memory per process.

4.1 Deterministic solution with identifiers

The intuition of the solution is as follows. Every process collects the predecessor information from all of its ancestors using the self-stabilizing approach given in [9, 12, 15]. From the collected information, each process i can reconstruct the exact topology of the subgraph consisting of all the ancestors and i itself. Then, depending on where the process is located, two possibilities can be considered:

1. The process is in a strongly connected component that includes all of its ancestors. In the directed acyclic graph of strongly connected components, this process is in a *source* component. Then every process in the source component constructs the same topology. The MIS in this source component is constructed for example by giving processes priority in the descending order of identifiers (*i.e.*, the process with maximal identifier has highest priority).

2. The process is in a *non-source* strongly connected component in the same acyclic graph of strongly connected components. Then, the same process as in the previous situation repeats, with the additional constraint that stronger priority is given to the processes in the upwards strongly connected components.

The detailed algorithm is given in Algorithm 4.1.

Algorithm 4.1 Deterministic UMIS algorithm in asynchronous networks with identifiers

```
constants of process i
    id_i: identifier of i;
    P_i: identifier set of P.i;
variables of process i
    Topology_i: set of (id, ID, dist) tuples; // topology that i is currently aware of.
        // id: a process identifier
        // ID: identifier set of P.(id)
        // dist: distance from id to i in Topology_i.
function
    update(Topology_i)
        Topology_i := {(id_i, P_i, 0)} ∪ ⋃_{j∈P.i} {(id, ID, dist + 1) | (id, ID, dist) ∈ Topology_j};
        while ∃(id, ID, dist), (id', ID', dist') ∈ Topology_i s.t. id = id' and dist < dist'
            remove (id', ID', dist') from Topology_i;
        while ∃(id, ID, dist), (id', ID', dist') ∈ Topology_i s.t. id = id' and ID ≠ ID'
            remove one of them (arbitrarily) from Topology_i;
        while ∃(id, ID, dist) ∈ Topology_i s.t. id is unreachable to i in Topology_i
            remove (id, ID, dist) from Topology_i;
    UMIS(Topology_i)
        WorkingTp_i := Topology_i;
        UMIS_i := ∅
        while ∃(id_i, P_i, 0) ∈ WorkingTp_i {
            Let W be a source strongly connected component of WorkingTp_i;
            for each id ∈ W in the descending order of identifiers
                if UMIS_i ∪ {id} is an independent set
                    UMIS_i := UMIS_i ∪ {id};
            WorkingTp_i := WorkingTp_i − W;
        }
        if id_i ∈ UMIS_i
            output true;
        else
            output false;
actions of process i
    true ⟶ update(Topology_i); UMIS(Topology_i);
```

Lemma 1 *Let i be any process. At the end of the k-th round ($k \geq 1$) and later, the topology stored in variable $Topology_i$ is correct up to distance $k-1$:*

1. *for every process j with $d(j,i) \leq k-1$, $Topology_i$ stores the correct tuple $(j, P.j, d(j,i))$ of j, and*
2. *every tuple $(id, ID, d) \in Topology_i$ is the correct one $(j, P.j, d(j,i))$ of some process j if $d \leq k-1$.*

Proof. We prove the lemma by induction on k. Let us observe first that the lemma holds for $k = 1$ (inductive basis): Once i executes its action, $Topology_i$ always contains $(i, P.i, 0)$ and any other tuple (id, ID, d) satisfies $d \geq 1$.

Assuming that the lemma holds for k (inductive hypothesis), we now prove the lemma for $k+1$ (inductive step). Any process u with $d(u,i) \leq k$ satisfies $d(u,j) = d(u,i) - 1 \leq k-1$ for some predecessor j of i. From the inductive hypothesis, $Topology_j$ contains the correct tuple $(u, P.u, d(u,j))$ of u at the end of the k-th round and later. Thus, i reads the correct tuple $(u, P.u, d(u,j))$ in $Topology_j$ and updates its distance correctly at every action in the $(k+1)$-th round and later. The hypothesis also implies that any tuple (u, ID, d) contained in $Topology_v$ of any predecessor v of i after the end of the k-th round satisfies $d \geq d(u,i) - 1$ and is correct if $d = d(u,i) - 1$. Thus, the correct tuple $(u, P.u, d(u,i))$ is never removed from $Topology_i$ in the $(k+1)$-th round or later. The first claim of the lemma holds for $k+1$.

Existence of tuple (id, ID, d) $(d \neq 0)$ in $Topology_i$ at the end of the $(k+1)$-th round or later implies that i reads $(id, ID, d-1)$ in $Topology_j$ of some predecessor j of i. From the hypothesis, any tuple $(id, ID, d-1)$ contained in $Topology_j$ after the end of the k-th round is *correct* (or id is an identifier of a really existing process, say v, ID is the identifier set of $P.v$ and $d = d(v,j)$ holds) if $d - 1 \leq k - 1$. Thus, any tuple (id, ID, d) contained in $Topology_i$ at the end of the $(k+1)$-th round or later is correct if $d \leq k$. The second claim of the lemma holds for $k+1$. □

The following corollary is derived from Lemma 1.

Corollary 1 *Let i be any process and $D(i)$ be the maximum distance to i from all the ancestors of i. At the end of the $(D(i)+1)$-th round and later, $Topology_i$ stores the exact topology of the subgraph consisting of all the ancestors of i and i itself.*

Proof. Concerning $Topology_i$ at the end of the $(D(i)+1)$-th round and later, Lemma 1 shows that the correct tuple $(u, P_u, d(u,i))$ of every ancestor u of i is contained, and any tuple (id, ID, d) with $d \leq D(i)$ is correct. This implies that $Topology_i$ at the end of the $(D(i)+1)$-th round and later can contain no tuple (id, ID, d) with $d > D(i)$ since the process with identifier id is not reachable to i in $Topology_i$ and such a tuple is removed from $Topology_i$ if exists. Thus the corollary holds. □

Theorem 3 *Algorithm 4.1 presents a self-stabilizing deterministic UMIS algorithm in asynchronous networks with identifiers. Its convergence time is $D+1$ rounds where D is the diameter of the network, and the memory space used at each node is $O(m \log n)$ bits.*

Proof. Let *Topology* be the exact topology of the network. It is obvious that *UMIS*(*Topology*) correctly finds a UMIS when executed until $WorkingTP = \emptyset$ holds. When $Topology_i$ stores the exact topology of the subgraph consisting of all ancestors of i, *UMIS*($Topology_i$) selects i as a member of UMIS iff *UMIS*(*Topology*) selects i: whether process i is selected by *UMIS*(*Topology*) depends only on the topology of the subgraph consisting of all ancestors of i. Corollary 1 guarantees that $Topology_i$ of every process i stores such exact topology at the end of the $(D+1)$-th round and later, and thus, the theorem holds. As the *Topology* variable may end up in containing an entry for every node, the over space needed is $O(m \log n)$ bits per process. □

Notice that Algorithm 4.1 enables each process i to know eventually the exact topology of the subgraph consisting of all the ancestors of i. Algorithm 4.1 can be easily extended so that each process can eventually get the exact topology containing the input values of the ancestors if each process has a *static* input value. Such an extension results in a *universal* scheme since it can solve any non-reactive problem that is consistently solvable at each process using the topology and the input values of its ancestors.

Another observation is that Algorithm 4.1 can easily be modified to become *silent*. For simplicity of our presentation, every process always has an enabled action with guard **true**, and thus, Algorithm 4.1 is not silent. But, Algorithm 4.1 becomes silent by changing the guard so that the action becomes enabled only when $Topology_i$ needs to be updated.

4.2 Probabilistic solution with unbounded memory in asynchronous anonymous networks

In this subsection, we present a probabilistic self-stabilizing UMIS algorithm for asynchronous *anonymous* networks. The solution is based on a probabilistic unique naming of the processes and a deterministic UMIS algorithm that assumes unique process identifiers. In the naming algorithm, each process is given a name variable that can be arbitrary large (thus the unbounded memory requirement). The naming is unique with probability 1 after a bounded number of new name draws. The new name draw consists in appending a random bit at the end of the current identifier. Each time the process is activated, a new random bit is appended. In parallel, we essentially run the deterministic UMIS algorithm presented in the previous subsection. The main difference from the previous algorithm is in handling the process identifiers. The variable *Topology* of a particular process may contain several different identifiers of a same process since the identifier of the process continues to get longer and longer in every execution of the protocol. To circumvent the problem, we consider two distinct identifiers to be the *same* if one is a prefix of the other, and anytime such same identifiers conflict, only the longest one is retained. Another difference is that we do not need the distance information. The distance information is used in the previous algorithm to remove the *fake* tuples (i, ID, d) of process i such that $ID \neq P.i$, which may exist in the initial configuration. In our scheme, tuples with fake identifiers that are prefixes of identifiers of real processes are eventually removed in Algorithm 4.2 since the

correct identifier eventually becomes longer than any fake identifier. Other tuples with fake identifiers are eventually disconnected from the constructed subgraph topology.

The details of the algorithm are given in Algorithm 4.2; only the topology update part is described since the UMIS function is the same as in Algorithm 4.1.

Algorithm 4.2 Probabilistic UMIS algorithm in asynchronous anonymous networks

```
variables of process i
    id_i: identifier (binary string) of i;
    P_i: identifier set of P.i;
    Topology_i: set of (id, ID) tuples; // topology that i is currently aware of.
        // id: a process identifier
        // ID: identifier set of P.(id)
function
    update(Topology_i)
        id_i := append(id_i, random_bit); // append a random bit to the current id
        Topology_i := {(id_i, P_i)} ∪ ⋃_{j∈P.i} Topology_j;
        while ∃(id, ID), (id', ID') ∈ Topology_i s.t. id' is a prefix of id
            remove (id', ID') from Topology_i;
        while ∃(id, ID) ∈ Topology_i s.t. id is unreachable to i in Topology_i
            remove (id, ID) from Topology_i;
```

Theorem 4 *Algorithm 4.2 presents a self-stabilizing probabilistic UMIS algorithm in asynchronous anonymous networks. Its expected convergence time is* $O(\log n + \log \ell + D)$ *rounds where* D *is the diameter of the network and* ℓ *is the number of fake identifiers in the initial configuration.*

Proof Sketch: It is clear that the identifier of any process eventually becomes distinct from any other's with probability 1. We first show that every process has a unique identifier in $O(\log n)$ expected rounds.

We consider, as the worst-case scenario, the case where all processes start with the same identifier and each process appends only a single bit to its identifier in every round.

The probability that every process has a unique identifier at the end of round k (*i.e.*, n random strings of k bits are mutually distinct) is evaluated as follows when n is small compared to 2^k:

$$\prod_{i=1}^{n-1}(1-\frac{i}{2^k}) \approx \prod_{i=1}^{n-1} exp(-\frac{i}{2^k}) = exp(-\frac{n(n-1)}{2^{k+1}}) \approx exp(-\frac{n^2}{2^{k+1}})$$

We introduce a discrete random variable X to represent the number of rounds required until every process has a unique identifier. When we consider the execution after round $2\log n$ to

guarantee n is small compared to 2^k, the expected number of rounds is then bounded by

$$2\log n + \sum_{i=2\log n}^{\infty} Prob(X > i) = 2\log n + \sum_{i=2\log n}^{\infty} (1 - exp(-\frac{n^2}{2^{i+1}}))$$

$$\approx 2\log n + \sum_{i=2\log n}^{\infty} \frac{n^2}{2^{i+1}} = 2\log n + O(1)$$

Thus, every process has a unique identifier in expected $O(\log n)$ rounds.

Processes may still have same identifiers as those contained in fake tuples. By a similar argument to the above, we can see additional $O(\log \ell)$ expected rounds are sufficient to give each process an identifier distinct from any fake one. Then, all the fake identifiers are removed from $Topology_i$ of each process i since such identifiers either become unreachable to i in $Topology_i$ or become prefixes of real indentifiers.

After all identifiers become distinct from one another, the topology stored in $Topology_i$ of each process i becomes stable if the process identifiers are ignored (*i.e.*, only process identifiers get longer and longer). On the other hand, once the identifier of a process u becomes lexicographically larger than that of a process v, u's identifier is lexicographically larger than v's afterward. This guarantees that every execution of $UMIS(Topology_i)$ at process i after some point returns the same result concerning whether process i is a member of the UMIS or not. By similar discussion to the proof of Theorem 3 we can show that additional $O(D)$ rounds are sufficient to get the stable UMIS solution once every process has a unique identifier.

Consequently, Algorithm 4.2 presents a self-stabilizing probabilistic UMIS algorithm and its expected convergence time is $O(\log n + \log \ell + D)$ rounds. □

4.3 Probabilistic solution with bounded memory in synchronous anonymous networks

The algorithm in the previous section is based on *global* unique naming, however, self-stabilizing global unique naming in unidirectional networks inherently requires *unbounded* memory. The goal of this subsection is to achieve, with *bounded* memory, a *local* unique naming that gives each process an identifier that is different from that of any of its ancestors, and to compute a UMIS based on the previously computed local naming. Indeed, such a local naming is sufficient for each process to recognize the strongly connected component it belongs to. Once the component is recognized, a UMIS can be computed by a method similar to that of the deterministic algorithm presented in Section 4.1.

In our scheme to achieve local unique naming, each process extends its identifier by appending a random bit when it finds an ancestor with the same identifier as its own. To be able to perform such a detection, a process needs to distinguish any of its ancestors from itself even when they have the same identifier. The detection mechanism is basically executed as follows: each process draws a random number, and disseminates its identifier

together with the random number to its descendants. When process i receives the same identifier as its own, it checks whether the attached random number is same as its own. If they are different, the process detects that this is a distinct process (that is, a real ancestor) with the same identifier as its own current identifier. When the process receives the same identifier with the same random number as its own for a given period of time, it draws a new random number and repeats the above procedure. Hence, as two different processes eventually draw different random numbers, eventually every process is able to detect an ancestor with the same identifier if such an ancestor exists.

The above method may cause *false detection* (or false positive) when a process receives its own identifier but with an old random number. To avoid such false detection, each identifier is relayed with a distance counter and is removed when the counter becomes sufficiently large. Moreover, the process repeats the detection checks while keeping sufficiently long periods of time between them. The details of the self-stabilizing probabilistic algorithm for the local naming are presented in Algorithm 4.3.

Algorithm 4.3 Probabilistic local naming in synchronous anonymous networks

```
variables of process i
    id_i: identifier (binary string) of i;
    rnd_i: random number selected from {1, 2, ..., k}; // k (≥ 2) is a constant
    ID_i: set of (id, rnd, dist) tuples; // identifiers that i is currently aware of.
        // id: a process identifier
        // rnd: random number of P.(id)
        // dist: distance that id traverses
function
    update(ID_i)
        ID_i := {(id_i, rnd_i, 0)} ∪ ⋃_{j∈P.i} {(id, rnd, dist + 1) | (id, rnd, dist) ∈ID_j};
        while ∃(id, rnd, dist) ∈ ID_i s.t. dist > |{id | (id, *, *) ∈ID_i}|;
            remove (id, rnd, dist) from ID_i;
        if timer > |{id | (id, *, *) ∈ID_i}| // timer is incremented by one every round
            naming(ID_i)
    naming(ID_i)
        if ∃(id_i, rnd, *) ∈ ID_i s.t. rnd ≠ rnd_i
            id_i := append(id_i, random_bit); // append a random bit to the current id
        rnd_i := number randomly selected from {1, 2, ..., k};
        reset_timer; // reset timer to 0
        update(ID_i);
actions of process i
    true ⟶ update(ID_i);
```

Lemma 2 *Algorithm 4.3 presents a self-stabilizing probabilistic local naming algorithm in synchronous anonymous networks. Its expected convergence time is $O((n+\ell)\log n)$ rounds where ℓ is the number of fake identifiers in the initial configuration.*

Proof sketch: First we show that the algorithm is a self-stabilizing probabilistic local naming algorithm. For contradiction, we assume that two processes i and j (where j is an ancestor of i) keep a same identifier after a configuration. Without loss of generality, the distance from j to i is minimum among process pairs keeping same identifiers. Let $j, u_1, u_2, \ldots, u_m, i$ be the shortest path from j to i. Since all processes in the path have mutually distinct identifiers except for a pair i and j, (id_j, rnd_j) is not discarded in the intermediate processes and is delivered to i. Thus, eventually i detects $id_i = id_j$ and $rnd_i \neq rnd_j$. Then i extends its identifier by adding a random bit, which is a contradiction.

We evaluate the expected convergence time of the algorithm. By similar argument to the proof of Theorem 4, we can show that the expected number of bits added to a process identifier is $O(\log n)$. Notice that the number ℓ of fake identifiers has no influence to the evaluation, for the distance *dist* of a fake identifier is larger than the timer value (once the timer is reset) and thus is removed (because of $dist > |\{id \mid (id, *, *) \in ID_i\}|$) when function *naming* is executed. On the other hand, in the scenario where all processes start with a same identifier, the time between two executions of function *naming* at a process is $O(n+\ell)$. Thus, the expected convergence time is $O((n+\ell)\log n)$ rounds. □

Algorithm 4.4 presents a self-stabilizing UMIS algorithm in locally-named networks. Thus, the *fair composition*[11] of the algorithm with the local-naming algorithm in Algorithm 4.3 provides a self-stabilizing UMIS algorithm in synchronous anonymous networks. For simplicity, we omit the code for removing fake initial information in Algorithm 4.4 since such fake initial information can be removed in a similar way to Algorithm 4.3.

Lemma 3 *In the algorithm presented in Algorithm 4.4, each process can exactly recognize the topology of the strongly connected component it belongs to in $O(D)$ rounds where D is the diameter of the network.*

Proof sketch: It is obvious that variable $Topology_i$ of each process i after D rounds consists of tuples $(id, P.(id))$ from all the ancestors of i . Notice that the local naming allows two distinct processes to have a same identifier if they are mutually unreachable. Thus, $Topology_i$ may contain a same tuple (id, P) of two or more distinct processes and/or may contain two tuples (id, P) and (id, P') with a same id but different predecessor sets P and P'.

Each process constructs the following graph $G_i = (V_i, E_i)$: $V_i = \{id \mid (id, *) \in Topology_i\}$ and $E_i = \{(u, v) \mid (v, P) \in Topology_i \text{ s.t. } u \in P\}$. In other words, G_i can be obtained from the actual graph G as follows: First consider the subgraph G'_i induced by the ancestors of i and i itself, and then merge the processes with the same identifier into a single process.

It is obvious that all processes in G_i are reachable to i. What we have to show is that process j is reachable from i in G_i (or j belongs to the strongly connected component of i) if and only if j is also reachable from i in G'_i. The if part is obvious since G_i is obtained from G'_i by merging processes. The only if part holds as follows. Consider two distinct processes j

Algorithm 4.4 UMIS algorithm in locally-named networks

```
constants of process i
    id_i: identifier of i; // distinct from that of any ancestor
    P_i: identifier set of P.i;
variables of process i
    umis_i: boolean; // true iff i is a UMIS node
    Topology_i: set of (id, ID) tuples; // topology that i is currently aware of.
        // id: a process identifier
        // ID: identifier set of P.(id)
    Comp_i: identifier set // of processes in the strongly-connected component of i
function
    update(Topology_i)
        Topology_i := {(id_i, P_i)} ∪ ⋃_{j∈P.i} Topology_j;
    UMIS(Topology_i)
        Comp_i := {id | id is reachable from i in Topology_i};
            // set of processes in the strongly connected component of i
        UMIS_v := ∅;
        if ∃j ∈ P.i− Comp_i s.t. umis_j = true or ∃j ∈ Comp_i s.t. (j > i and umis_j = true) {
            umis_i := false; output false;
        }
        else {
            umis_i := true; output true;
        }
actions of process i
    true ⟶ update(Topology_i); UMIS(Topology_i);
```

and j' with a same identifier if exist. Since they are mutually unreachable but are reachable to i, they are unreachable from i in G'_i (otherwise one of them is reachable from the other). In construction of G_i from G'_i, merging is applied only to processes *unreachable from* i, that is, the merging has no influence on reachability *from* i. Thus, any process unreachable from i in G'_i remains unreachable from i in G_i. □

Lemma 4 *Algorithm presented in Algorithm 4.4 is a self-stabilizing (deterministic) UMIS algorithm in (asynchronous) locally-named networks. Its convergence time is $O(n)$ rounds.*

Proof sketch: First from Lemma 3, every process correctly recognizes in $O(D)$ rounds all the processes in the same connected component. Then consider a *source* strongly connected component. The process with the maximum identifier in the component becomes a *stable* UMIS member. After that the UMIS outputs of processes in the component become stable one by one in the descending order of identifiers. It takes at most $O(n')$ rounds until all

the processes in the component become stable, where n' is the number of processes in the component.

The same argument can be applied to a source strongly connected component in the graph obtained from G by removing the components with stabilized UMIS outputs. By repeating the argument, we can show that the UMIS outputs of all the processes become stable in $O(n)$ rounds. It is clear that the processes with the UMIS outputs of **true** form a UMIS. □

From Lemmas 2 and 4, the following theorem holds.

Theorem 5 *Fair composition of algorithms presented in Algorithm 4.3 and Algorithm 4.4 provides a self-stabilizing probabilistic UMIS algorithm in synchronous anonymous networks. Its expected convergence time is $O((n+\ell)\log n)$ rounds where ℓ is the number of fake identifiers in the initial configuration. The space complexity of the resulting protocol is $O(n \log n)$.*

5 Conclusion

Although in bidirectionnal networks, self-stabilizing maximal independent set is as difficult as vertex coloring [16], this work proves that in unidirectionnal networks, the computing power and memory that is required to solve the problem varies greatly. Silent solutions to unidirectional uniform networks coloring require $\Theta(\log n)$ (resp. $\Theta(\log \delta)$, where δ denotes the maximal degree of the communication graph) bits per process and have stabilization time $\Theta(n^2)$ (resp. $\Theta(1)$) when deterministic (resp. probabilistic) solutions are considered. By contrast, deterministic maximal independent set construction in uniform networks is *impossible*, and silent maximal independent set construction is *impossible*, regardless of the deterministic or probabilistic nature of the protocols.

While we presented positive results for the deterministic case with identifiers, and the non-silent probabilistic cases, there remains the immediate open question of the possibility to devise a probabilistic solution with bounded memory in asynchronous setting.

Another interesting issue for further research related to *global* tasks. The global unique naming that we present in section 4.2 solves a truly global problem in networks where global communication is *not* feasible, by defining proper equivalences classes between various identifiers. The case of other classical global tasks in distributed systems (*e.g.* leader election) is worth investigating.

References

[1] Yehuda Afek and Anat Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.

[2] Yehuda Afek and Shlomi Dolev. Local stabilizer. *J. Parallel Distrib. Comput.*, 62(5):745–765, 2002.

[3] Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. *Distributed Computing*, 20(1):39–51, 2007.

[4] Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Bounds for self-stabilization in unidirectional networks. Technical report, INRIA, May 2008.

[5] Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *Proceedings of the IEEE International Conference on Parallel and Distributed Processing Systems (IPDPS 2009)*, Rome, Italy, May 2009. IEEE Press.

[6] Jorge Arturo Cobb and Mohamed G. Gouda. Stabilization of routing in directed networks. In Ajoy Kumar Datta and Ted Herman, editors, *WSS*, volume 2194 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2001.

[7] Sajal K. Das, Ajoy Kumar Datta, and Sébastien Tixeuil. Self-stabilizing algorithms in dag structured networks. *Parallel Processing Letters*, 9(4):563–574, December 1999.

[8] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication*, 2006.

[9] Sylvie Delaët and Sébastien Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing*, 62(5):961–981, May 2002.

[10] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[11] S. Dolev. *Self-stabilization*. MIT Press, March 2000.

[12] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[13] Shlomi Dolev and Elad Schiller. Self-stabilizing group communication in directed networks. *Acta Inf.*, 40(9):609–636, 2004.

[14] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, July 2001.

[15] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science*, 293(1):219–236, 2003. Extended abstract in Sirocco 2000.

[16] Maria Gradinariu and Sébastien Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *International Conference on Principles of Distributed Systems (OPODIS'2000)*, pages 55–70, Paris, France, December 2000.

[17] Toshimitsu Masuzawa and Sébastien Tixeuil. Bounding the impact of unbounded attacks in stabilization. In Ajoy Kumar Datta and Maria Gradinariu, editors, *SSS*, volume 4280 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006.

[18] Toshimitsu Masuzawa and Sébastien Tixeuil. Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. *International Journal of Principles and Applications of Information Science and Technology (PAIST)*, 1(1):1–13, December 2007.

[19] Nathalie Mitton, Eric Fleury, Isabelle Guérin-Lassous, Bruno Séricola, and Sébastien Tixeuil. On fast randomized colorings in sensor networks. In *Proceedings of ICPADS 2006*, pages 31–38. IEEE Press, July 2006.

[20] Mikhail Nesterenko and Anish Arora. Tolerance to unbounded byzantine faults. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, page 22. IEEE Computer Society, 2002.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399